May 2014

Accelerating Pattern Matching in Neuromorphic Text Recognition System Using Intel Xeon Phi Coprocessor

Khadeer Ahmed
Syracuse University

Abstract

Neuromorphic computing systems refer to the computing architecture inspired by the working mechanism of human brains. The rapidly reducing cost and increasing performance of state-of-the-art computing hardware allows large-scale implementation of machine intelligence models with neuromorphic architectures and opens the opportunity for new applications. One such computing hardware is Intel Xeon Phi coprocessor, which delivers over a TeraFLOP of computing power with 61 integrated processing cores. How to efficiently harness such computing power to achieve real time decision and cognition is one of the key design considerations. This work presents an optimized implementation of Brain-State-in-a-Box (BSB) neural network model on the Xeon Phi coprocessor for pattern matching in the context of intelligent text recognition of noisy document images. From a scalability standpoint on a High Performance Computing (HPC) platform we show that efficient workload partitioning and resource management can double the performance of this many-core architecture for neuromorphic applications.

Accelerating Pattern Matching in Neuromorphic Text Recognition System Using Intel Xeon Phi Coprocessor

by

Khadeer Ahmed

B.E., Visvesvaraya Technological University, 2006

Thesis
Submitted in partial fulfillment of the requirements for the degree of
Masters of Science Computer Engineering.

Syracuse University
May 2014

Acknowledgement

I would like to thank my advisor Dr. Qinru Qiu for providing me an opportunity and supporting me in my efforts for this work. She provided me a platform to launch and was always available to guide me with her wisdom.

Also would like to acknowledge the help and invaluable guidance of Dr. Parth Malani and Mangesh Tamhankar from Intel Corporation, who facilitated this as an internship project. Parth was my mentor at Intel who helped in providing technical insights and project development.

I am great full to everyone who facilitated my work by helping me gain valuable experience into cutting edge, upcoming technologies and for the industry exposure.

Finally would like to thank my friends and in particular my family for their encouragement, support and unwavering confidence in me.

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

1.1 Overview

Cognitive computing is an emerging field made possible due to the advancement in High Performance Computing (HPC) domain. There is a special interest in cognitive computing because the challenges which are being faced today have no definite way of deriving a solution and hence cannot be coded in the traditional style. There are many examples which fall under this category, including natural language understanding, text image recognition, autonomous unmanned vehicle controls, user preference suggestion, etc.

Neuromorphic computing systems refer to the computing architecture inspired by the working mechanism and massive parallel structure of human brains. A neuromorphic information processing model is presented in [1]. The model consists of a simple but massive parallel pattern matching layer that generates fuzzy results retaining rich information (sometimes referred as ambiguity) and a powerful information inference layer that removes the ambiguity by statistical inference. The event-driven computation in neuromorphic engines loosely represents the integrate-and-fire behavior of neurons, leading to high computation efficiency. The hierarchical architecture mimics the primary sensory cortex and association cortex in brain's sensory processing area [2][3][4]. The model is applied for intelligent text recognition in document image processing, where meaningful sentences are extracted from camera captured documents and road signs. These are non-trivial problems as the images are captured with perspective distortion, angular distortion, warping or other general noises. Reliable

performance is achieved even when the system is exposed to new experiences. As their experience gets richer and richer for every new exposure, their performance improves.

The state-of-the-art multi-core computer architecture enables large-scale implementation of neuromorphic models. One of such computer system is Intel's Xeon Phi coprocessor, where more than 61 X86 compatible cores with 4-way multi-threading capability are integrated on a single die. Each core has its own L1 and L2 cache and can access coherent L2 caches of any other core [5]. This architecture delivers over a TeraFLOP computing bandwidth. How to harness such computing power to achieve real-time cognition and decision is an urgent research problem.

In this work, we accelerate the performance of the neuromorphic model for *Intelligent Text Recognition System* (ITRS) using the hybrid Xeon - Xeon Phi coprocessor setup. According to Amdahl's law, the most effort in performance optimization should be spent on the most common operation, which is identified to be pattern matching in ITRS. Pattern matching forms the bottom layer of ITRS. It is implemented using an auto-associative memory model called *Brain-State-in-a-Box* (BSB). BSB is a simple nonlinear, energy minimizing neural network [7][8][9], whose convergence speed is proportional to the similarity between the input image and the stored pattern. The bottom layer of the ITRS consists of large number of independent BSB models, which are ideal candidates for parallel implementation.

A scalable platform capable of handling images with different resolutions for higher fidelity pattern matching is developed on Intel's Xeon Phi coprocessor. One of the reasons to select Xeon Phi over GPGPU is that the applications can run natively on Xeon Phi compared to the

offload model of GPGPU. This is particularly an important capability as it frees up the CPU to handle more control intensive functions of ITRS [1].

1.2 RELATED WORK

The first stage of a text recognition system is image processing, which feeds the Optical Character Recognition (OCR) modules. A wide variety of OCRs for printed text recognition were developed over the years [10]. There is a heavy focus on improving their accuracy by employing various image processing and classifying techniques including neural network based learning techniques[11][12][13]. In ITRS, the intelligence for recognition and error correction has been pushed to upper layers, where word and sentence contexts are considered. Hence it affords a simple fuzzy pattern-matching layer, which generates more than one likely matches for a given pattern. This is also why performance optimization rather than accuracy optimization is focused in this work.

There are prior attempts to implement the pattern matching layer of ITRS on IBM Cell processors [14][15]. However, the limited size of on-chip memory of Cell processor prevents us from processing higher resolution images in reasonable time. Attempts have been made to have a hardware solution for processing these BSB neural networks using memristor crossbar arrays [16]. However, no real hardware has been implemented yet. GPGPU based optimizations have been implemented for several similar neural network models [17][18][19]. They need special attention from the CPU to move computation data from main memory to GPU memory. Furthermore, combining CUDA with Message Passing Interface (MPI) is not an ideal option [20].

Efficient use of Single Instruction Multiple Data (SIMD) architecture is the key to achieving high performance with Xeon Phi coprocessor. Several SIMD vectorization techniques are proposed in [21]. A comprehensive guide on optimization for the coprocessor is given by [22][23]. Work on optimizing a data intensive application running natively on the coprocessor is presented in [24]. Its authors have tried to reduce inter-thread dependency to minimize thread syncing overhead. Efficient parallelization of batch pattern training algorithm for the coprocessor and other HPC platforms is proposed in [25]. It uses MPI to perform communication and reduction operations at the same time.

2 BACKGROUND

In this chapter, the computation model and software architecture of ITRS are presented, followed by a brief introduction of hardware architecture of Xeon Phi coprocessor.

2.1 Brain-state-in-a-box model

BSB model is a simple, auto-associative, nonlinear, energy minimizing neural network [7][8][9]. A common application of the BSB model is to recognize a pattern from the given noisy input. It can also be used as a pattern recognizer that employs a smooth nearness measure and generates smooth decision boundaries.

There are two main operations in a BSB model, Training and Recall. Here the focus is on BSB recall operation. The mathematical model of a BSB recall operation can be represented in the following form:

$$x(t+1) = S(\alpha * A * x(t) + \lambda * x(t) + \gamma * x(0)) \quad (1)$$

Where:

- x is an N dimensional real vector
- A is an NxN connection matrix
- $A * x\ (t)$ is a matrix-vector multiplication operation
- α is a scalar constant feedback factor
- λ is an inhibition decay constant
- γ is a nonzero constant if there is a need to maintain the input stimulation
- $S\ ()$ is the "squash" function defined as follows:

$$S(y) = \begin{cases} 1 & if \quad y > 1 \\ y & if \quad -1 \le y \le 1 \\ -1 & if \quad y < 1 \end{cases} \quad (2)$$

Note that in our implementation, we choose λ to be 1.0 and γ to be 0.0. But they can be easily changed to other values. Given any input pattern $x(0)$, the recall process executes equation (1) iteratively to reach convergence. A recall converges when all entries of $x(t+1)$ are either "1.0" or "-1.0".

The BSB model is selected in the ITRS for two reasons. First, it is simple to operate compared to other complex neural network models [4]. Although it has lower accuracy, the error can be corrected by the upper layer information processing. Second, its convergence roughly indicates the similarity between the input and the stored pattern. It is pointed out by [4] that the average convergence time of the BSB model increases as the input goes further away from the attractor. Such property enables the racing behavior in character recognition, which is discussed in confabulation sub section.

2.2 Cogent confabulation

Cogent confabulation is a connection-based cognitive computing model. It captures correlations between objects (or features) at the symbolic level and stores this information as a knowledge base [1]. Given an observation, familiar information with high relevancy will be recalled from the knowledge base.

Based on the theory, the cognitive information process consists of two steps: learning and recall. During learning, the knowledge links are established and strengthened as symbols are co-activated. During recall, a neuron receives excitations from other activated neurons. A "winner-takes-all" strategy takes place within each lexicon. Only the neurons (in a lexicon) that represent the winning symbol will be activated and the winner neurons will activate other

neurons through knowledge links. At the same time, those neurons that did not win in this procedure will be suppressed.

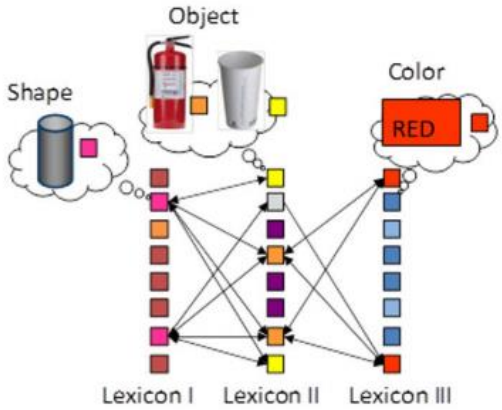


Figure 1: A simple example of lexicons, symbols, and knowledge links

Figure 1 shows an example of lexicons, symbols, and knowledge links. The three columns in Figure 1 represent three lexicons for the concept of shape, object, and color with each box representing a neuron. Different combinations of neurons represent different symbols. For example, the pink neurons in lexicon I represent the cylinder shape, the orange and yellow neurons in lexicon II represent a fire extinguisher and a cup, while the red neurons in lexicon III represent the red color. When a cylinder shaped object is perceived, the neurons that represent the concepts “fire extinguisher” and “cup” will be excited. However, if a cylinder shape and a red color are both perceived, the neurons associated with “fire extinguisher” receive more excitation and become activated while the neurons associated with the concept “cup” will be suppressed. At the same time, the neurons associated with “fire extinguisher” will further excite the neurons associated with its corresponding shape and color and eventually make those symbols stand out from other symbols in lexicons I and III.

2.3 Software Architecture of ITRS

ITRS is developed to extract meaningful sentences from document images. It has two information processing layers, a BSB model based pattern-matching layer and a confabulation model based statistical inference layer. The salient feature of ITRS is that it provides contextually correct sentence reconstruction even if there are illegible characters or words in the document image. This is enabled by a trained knowledge base, which captures the statistical information among building components in English language, from letters and words to phrases and part-of-speech tagging.

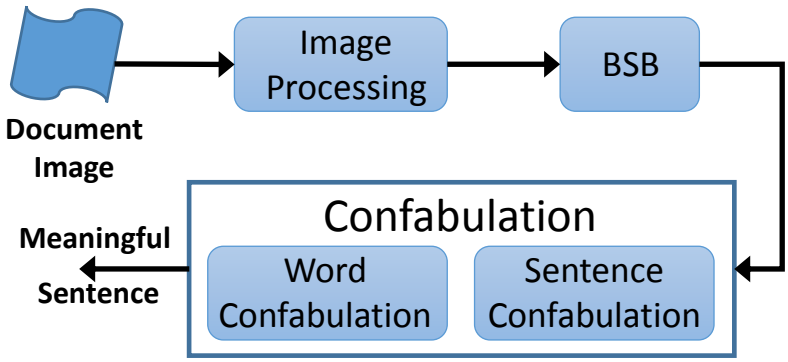


Figure 2: ITRS Pipeline

The information processing in ITRS has several stages, which can be arranged as a pipeline shown in Figure 2. After simple image processing which corrects image distortion, skew and warping, the character images are segmented from document image and forwarded to BSB, where fuzzy pattern matching is performed. The pattern matching results will be processed by word level and sentence confabulation for inference based error correction and association.

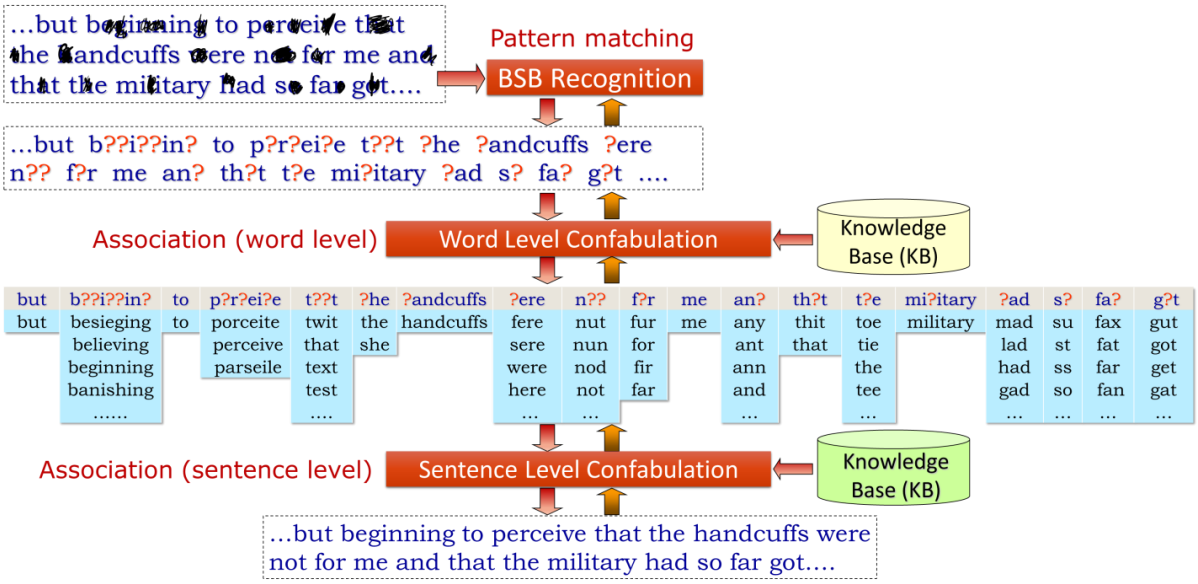


Figure 3: ITRS Cognitive model

A working example of ITRS is shown in Figure 3. Given a noisy document image, the BSB provides best effort pattern matching for each character images. Each question mark in the figure represents all 26 possible alphabets. The word confabulation layer forms all possible words based on these matching alphabets and the sentence confabulation layer selects the words that forms the most meaningful sentence. It is easy to see that, for each sentence, one sentence confabulation task and multiple word confabulation tasks must be executed, along with even more number of BSB pattern matching tasks. Information is passed across layers. The computation tasks in the same layer are independent to each other and hence can be implemented in parallel.

The pattern-matching layer (BSB) is trained on clean font image. The word level confabulation is trained by reading a dictionary. And the size of word level knowledge base is about 200 MB. The sentence level knowledge base is trained by reading multiple classic literatures. The size of this knowledge base is 6 - 12 GB.

3 Intel Xeon Phi coprocessor

3.1 Overview

Intel Xeon Phi coprocessor is based on Intel Many Integrated Core Architecture or Intel MIC. It is an architecture which consists of more than 50, modified x86 cores on a single die. The primary advantage of this architecture is that it can run the existing x86 code with very few modifications, at the same time get huge performance boost due to many independent parallel cores. The following sections briefly describe the hardware architecture of Knights Corner (KNC) generation Intel Xeon Phi coprocessor.

3.2 Micro architecture

The coprocessor primarily consists of processing cores, caches, memory controllers, PCIe client logic, and a very high bandwidth bidirectional ring interconnect as shown in Figure 4. Each core has a private L2 cache that is kept fully coherent by a global-distributed tag directory (TD). The memory controllers and the PCIe client logic provide a direct interface to the GDDR5 memory on the coprocessor and the PCIe bus, respectively. There are 8 memory controllers supporting up to 16 GDDR5 channels delivering up to 5.5 GT/s transfer speed, this provides a theoretical aggregate bandwidth of 352 GB/s (gigabytes per second) directly connected to the Intel® Xeon Phi™ coprocessor [6]. All these components are connected together by the ring interconnect.

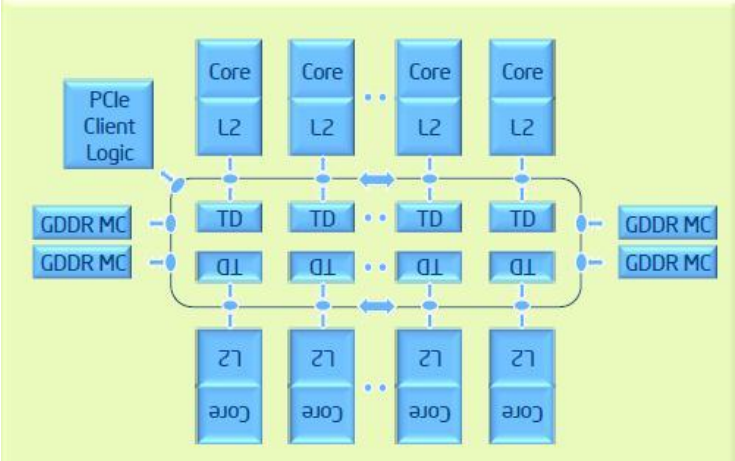


Figure 4: Xeon Phi microarchitecture

Source: http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

3.3 Core architecture

Each core in the coprocessor is power efficient and provides a high throughput for highly parallel workloads. It uses a short in-order pipeline and is capable of supporting 4 threads in hardware. It supports legacy IA (Intel Architecture) with an overhead of 2% with respect to the area costs of the core and is even less at the chip level [5]. The core architecture is as shown in Figure 5. The core can execute 2 instructions per clock cycle, one on the U-pipe and one on the V-pipe. The V-pipe cannot execute all instruction types, and simultaneous execution is governed by pairing rules. Most integer and mask instructions have a 1-clock latency, while most vector instructions have 4-clock latency with a 1 clock throughput. The Level One (L1) cache accommodates higher working set requirements for four hardware contexts per core. It has a 32 KB L1 instruction cache and 32 KB L1 data cache. It has 8-way associativity, with a 64 byte cache line. It also features a 512 KB unified Level Two (L2) cache. The L2 organization comprises 64 bytes per way with 8-way associativity [6].

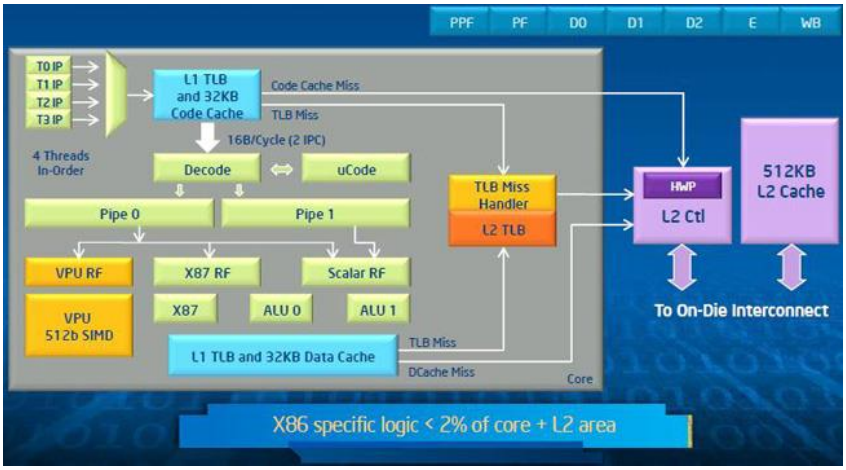


Figure 5: Xeon Phi core architecture

Source: http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

3.4 Vector Processing Unit

The vector processing unit (VPU) shown in Figure 6 is an important part of the core, which helps to greatly improve the performance. The VPU features a novel 512-bit SIMD instruction set, known as Intel Initial Many Core Instructions (Intel IMCI). The VPU can execute 16 single-precision (SP) or 8 double-precision (DP) operations per cycle. The VPU also supports Fused Multiply-Add (FMA) instructions and hence can execute 32 SP or 16 DP floating point operations per cycle. It also provides support for integers. The VPU contains the vector register file (32 registers per thread context), and can read one of its operands directly from memory, including data format conversion on the fly. Broadcast and swizzle instructions are also available [6].

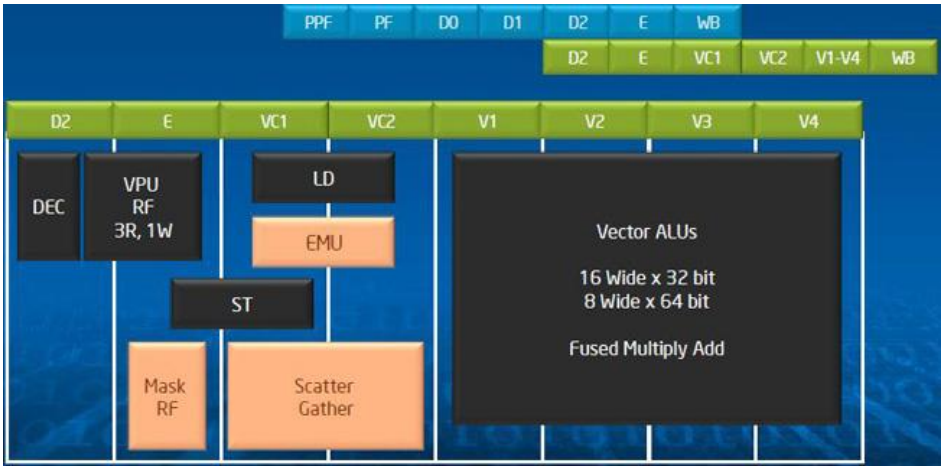


Figure 6: Vector Processing Unit

Source: http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

Vector units are very power efficient for HPC workloads. A single operation can encode a great deal of work and does not incur energy costs associated with fetching, decoding, and retiring many instructions. However, several improvements were made to support such wide SIMD instructions. For example, a mask register is added to the VPU to allow per lane predicated execution. This helps in vectorizing short conditional branches, thereby improving the overall software pipelining efficiency. The VPU also supports gather and scatter instructions, which are simply non-unit stride vector memory accesses, directly in hardware. Thus for codes with sporadic or irregular access patterns, vector scatter and gather instructions help in keeping the code vectorized.

The VPU also features an Extended Math Unit (EMU) that can execute transcendental operations such as reciprocal, square root, and log, thereby allowing these operations to be executed in a vector fashion with high bandwidth. The EMU operates by calculating polynomial approximations of these functions.

3.5 The Interconnect

The interconnect, as shown in Figure 7 is implemented as a bidirectional ring. Each direction is comprised of three independent rings. The first, largest, and most expensive of these is the data block ring. The data block ring is 64 bytes wide to support the high bandwidth requirement due to the large number of cores. The address ring is much smaller and is used to send read/write commands and memory addresses. Finally, the smallest ring and the least expensive ring is the acknowledgement ring, which sends flow control and coherence messages.

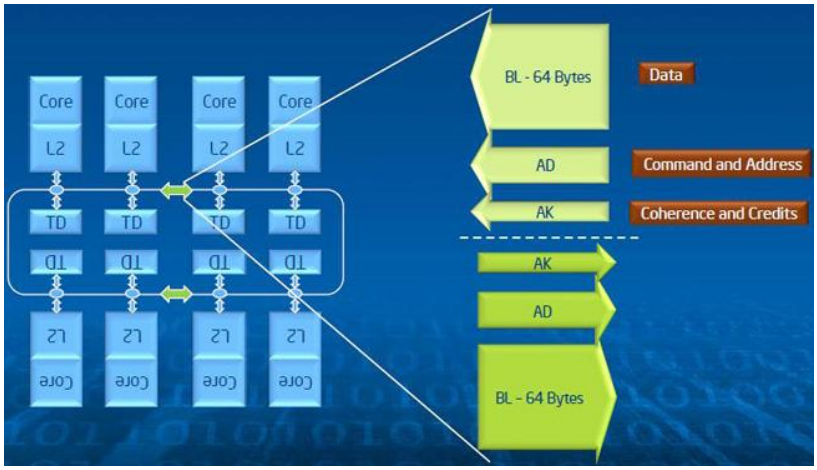


Figure 7: Interconnect

Source: http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

4 Implementation Overview

Sequential implementation of the ITRS software was used as the starting point, followed by 3 development phases. The first phase is to restructure the ITRS software to support multiple parallel BSB threads, and thus to parallelize the pattern matching layer. The BSB threads and the ITRS software were first implemented on a standalone Xeon host. Each BSB can handle 2 character image sizes, with 15x15 or 30x30 pixels per character image. They correspond to 256 bytes wide and 1024 bytes wide input vectors respectively. This is a crucial step as both Xeon host and the coprocessor are based on similar x86 architecture, hence initial code development and testing becomes smoother. Once the initial code is tested on Xeon host, it can also be compiled and run on Xeon Phi.

Naturally the next phase is to move the BSB code to Xeon Phi coprocessor, and port the image processing and confabulation models on the host Xeon processor of our system. Message Passing Interface (MPI) is used for data communication between confabulation and BSB. In this way the pattern-matching layer is separated from the rest of the ITRS and allocate more computing resources to it. Image processing module groups the character images in to workloads and issues them to BSB through MPI. Each workload consists of 96 characters, which is an input to the BSB module. The size is chosen to provide a good balance between communication and computation time. BSB compares each image in the workload against 93 trained patterns. The pattern set consists of lower case and upper case English alphabets, numbers and some common symbols & punctuations. The output is the set of matching patterns, which are called letter candidates, and their convergence speed. These results are used by the confabulation module to generate meaningful sentences.

The third and final phase is to optimize the BSB code for Xeon Phi coprocessor. Figure 8 shows the hybrid Xeon – Xeon Phi architecture for this phase. Confabulation is moderately parallel and is a tree structured algorithm. While BSB is massively parallel in nature. Xeon host also manages dispatching the images to BSB and collecting the results. The methods utilized and the avenues explored for optimization along with the development details are described in the following chapters.

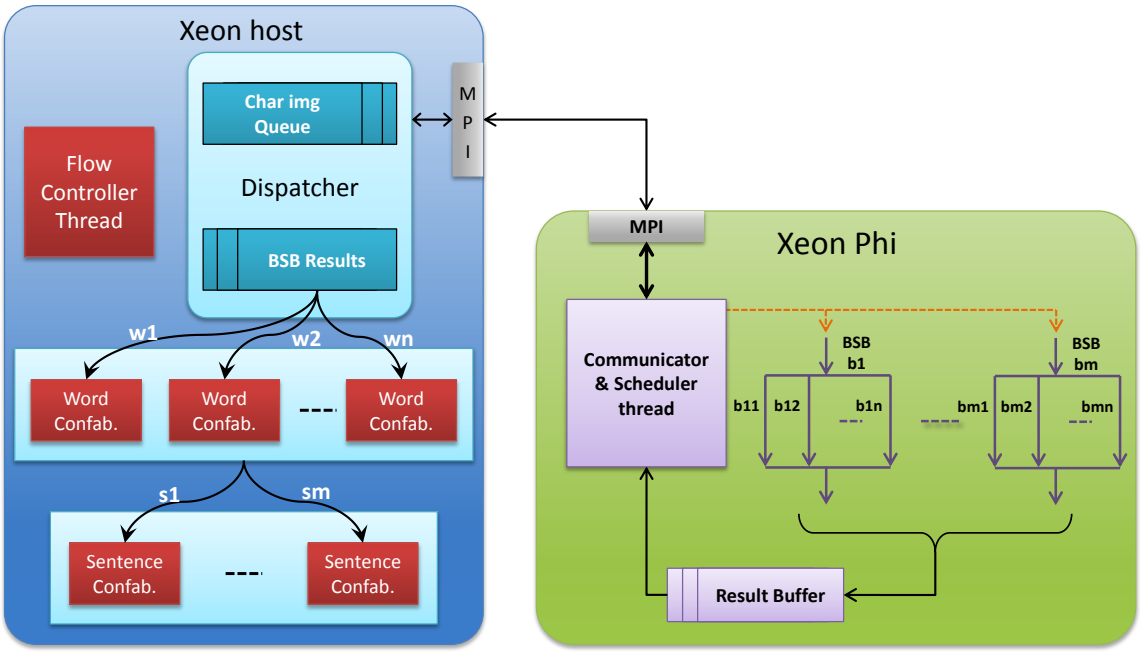


Figure 8: Software Architecture

4.1 System setup

An Intel 2-socket Xeon server host machine was used for the development. OpenMP, Intel MPI along with Intel C++ compiler (version: 13.0.5.192) were used and Intel VTune Amplifier was used to profile the code on both Xeon processor and the coprocessor. A micro OS runs on each Xeon Phi card known as Manycore Platform Software Stack (MPSS). Table shown below provides more details on the system setup.

Table 1: Setup details

	Xeon	Xeon Phi
Part number	E5-2687W	7110P
#Physical Processors	2	2
#Cores	8 (Total 8*2 = 16 cores)	61 (Total 61*2 = 122 cores)
Cache	L1 – 512KB L2 – 2MB L3 – 20MB	L1 – 32KB L2 – 512KB
#Threads	16*2 = 32	61*4 = 244
Frequency	3.1GHz	1.1 GHz
RAM	32 GB	GDDR 8 GB
OS	Redhat Enterprise 6.3 (2.6.32-279)	MPSS 2.1.5889-16

5 Phase 1 Experiments

In this phase the base line performance of serial code on Xeon host is determined and the BSB code is restructured for parallel implementation.

5.1.1 Experiment 1 – Serial code

Standalone serial BSB code was compiled using gcc and icc compilers and their runtimes compared as the existing Cell processor code is compiled using gcc compiler. This experiment is performed to judge the baseline performance of the initial setup. The results for 15x15 characters and 30x30 characters are shown in Figure 9 and Figure 10 respectively. From the charts it is evident that icc provides better performance compared to gcc with O3 level optimizations enabled.

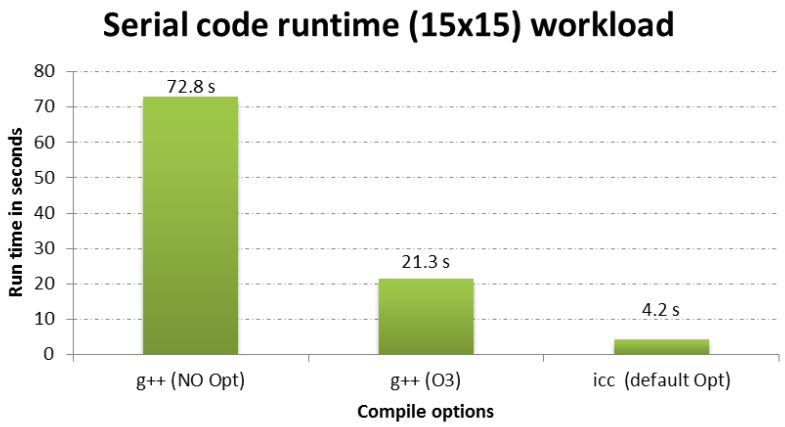


Figure 9: Baseline runtime for serial code in for 15x15 characters

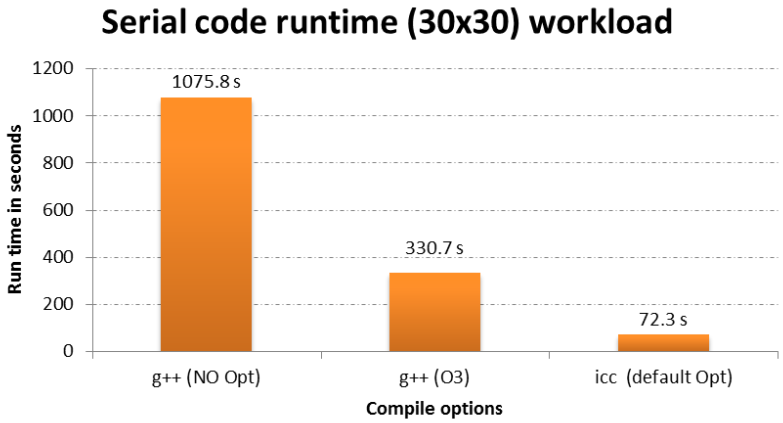


Figure 10: Baseline runtime for serial code in for 30x30 characters

5.1.2 Experiment 2 - OpenMP

The standalone Xeon BSB is parallelized using OpenMP and compiled using both gcc and icc for multicore CPU and the results are as shown in Figure 11 and Figure 12. Two kinds of parallelization were implemented using OpenMP: first with an OpenMP thread for every character in workload, second was with OpenMP task for every core compute section. For remainder of the work, all compilations are done using icc compiler.

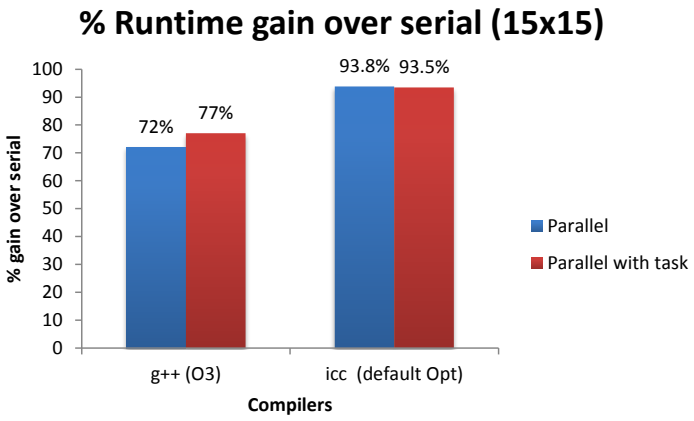


Figure 11: Runtime gain with OpenMP over serial for 15x15 case

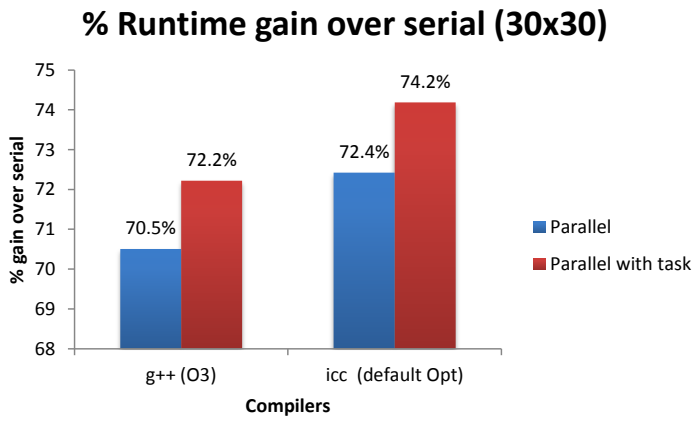


Figure 12: Runtime gain with OpenMP over serial for 30x30 case

5.1.3 Initial Runtime of MIC v/s Multicore CPU

In this experiment the BSB code was implemented on a multicore CPU architecture and Xeon OpenMP code was ported to the MIC architecture. From Figure 13 and Figure 14 it is clear that 15x15 case is suitable for multicore CPU and 30x30 case is suitable for MIC. Figure 15 shows that there is a drop in performance of for 15x15 case and gain for the case of 30x30 character. These results are for un-optimized code. After optimization there is performance gain for both the cases and these details are presented in chapter 7 and 8.

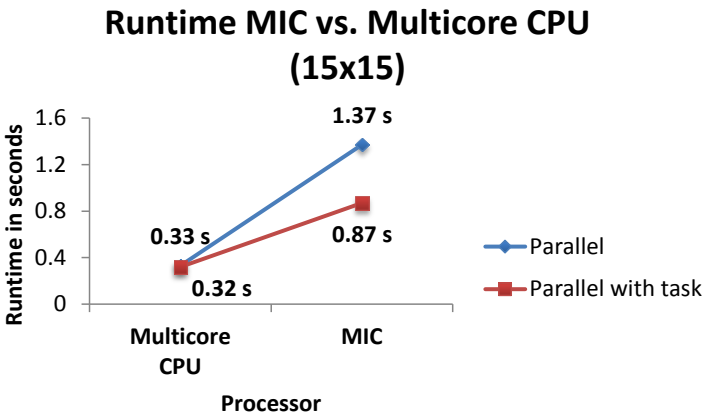


Figure 13: MIC vs multicore CPU runtime for 15x15 case

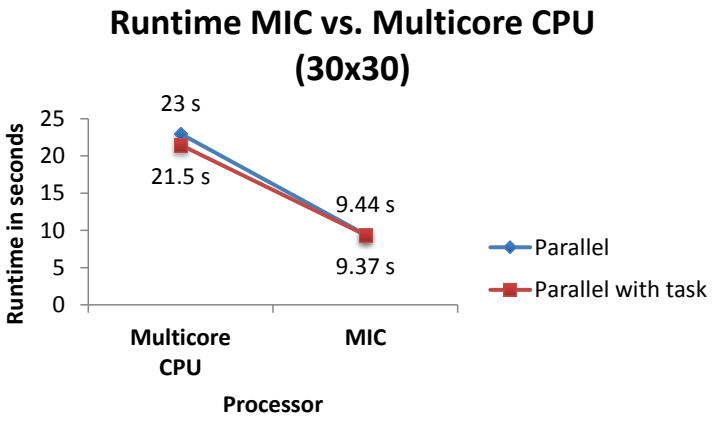


Figure 14: MIC vs multicore CPU runtime for 30x30 case

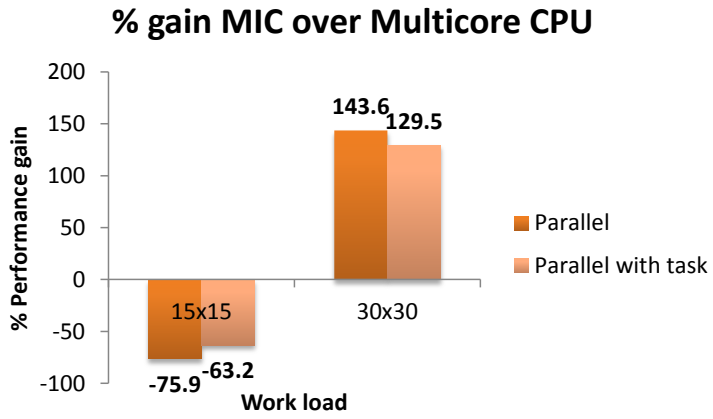


Figure 15: Performance gain of MIC over multicore CPU

The data requirement for 15x15 case is very low compared to 30x30 case and per core cache on multicore CPU is large compared to per core cache of MIC. Hence the parallelization overhead becomes significantly greater for 15x15 case on MIC. These results are not representative of the nature of the algorithm and platform as the code is not optimized yet. The results after optimization are presented in chapters 7 and 8.

6 Phase-2 Including confabulation

In this phase the BSB is interfaced with confabulation using MPI. The BSB code is compiled for Xeon Phi coprocessor. The image processing and confabulation models run on the host Xeon processor of our system. Experiments are carried out to evaluate the performance and the communication overhead.

6.1 Experiment

A higher resolution character image is preferred by the ITRS application hence, from this point onwards more concentration is given on 30x30 cases. All the experiments described in this section are for the case of confabulation integrated with un-optimized BSB. The initial BSB runtime on MIC was 18.41 seconds, some minor tweaking (reducing unnecessary computation) was performed to reduce the runtime. Figure 16 shows the average run time on MIC and multicore CPU with the error bars indicating the maximum and minimum runtimes obtained during this run.

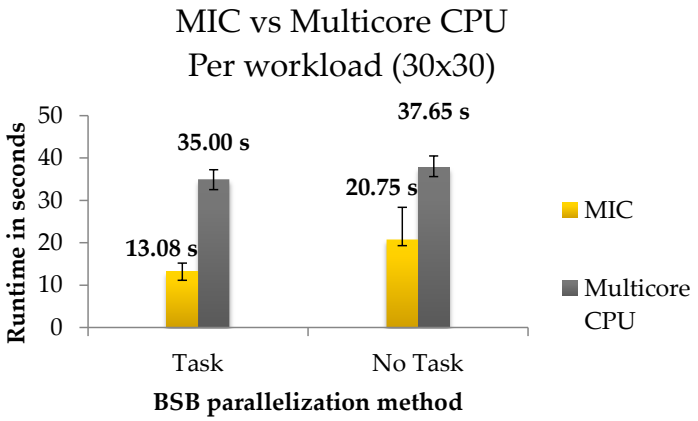


Figure 16: Runtime comparison of MIC vs multicore CPU with confab

The threads performing computation run independently from the MPI communication threads. Hence there is overlap of computation and communication, which is the ideal case as the MIC

spends maximum time possible for computation. Figure 17 shows the time spent on MPI communication with both receive and transmit time included for one workload. The MIC runtime of 13.08 seconds from Figure 16 mostly comprises of computation time. The corresponding MPI communication time from Figure 17 is 5.04 seconds.

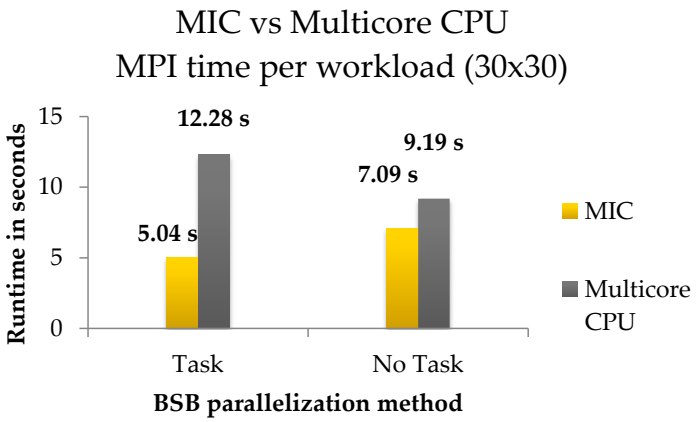


Figure 17: Time spent on MPI communication, MIC vs multicore CPU

Figure 18 summarizes the performance gain of MIC over multicore CPU due to drop in run time. With the un-optimized BSB code, gain of 167% was achieved with minor tweaking and just by porting the BSB to MIC.

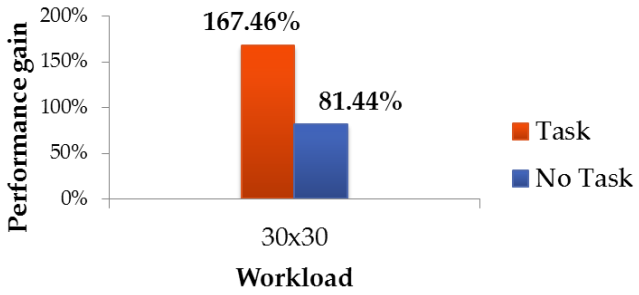


Figure 18: Performance gain of MIC over multicore CPU

The total time spent on the core compute part of the code is summarized in Figure 19. It is clear that MIC spends longer in the core compute section and one of the primary reason is, MIC runs

at 1GHz and multicore CPU at 3GHz. Even with this huge difference the MIC outperforms a multicore CPU as MIC is highly parallel.

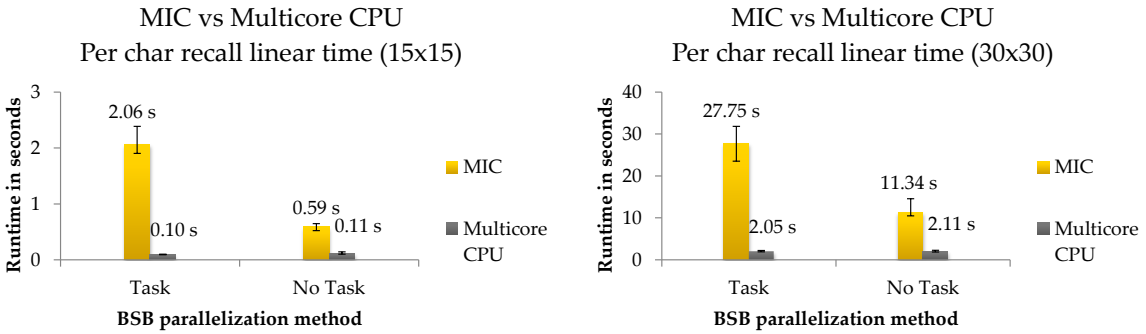


Figure 19: Total time spent on one character

6.2 Analysis

From the above experiment it can be inferred that, the workload is highly data intensive but benefits from parallelism. Though CPU is better in data handling, MIC has better overall runtime. Based on this inference the following optimization opportunities can be explored to improve overall performance:

- Memory pre-allocation on MIC
- Data localization
- Reduce per char runtime on MIC to increase performance
- Other MIC specific optimizations

7 Phase-3 Optimizations

Two progressive steps are taken to optimize the pattern-matching layer to exploit the resource on Xeon Phi. The first step is to restructure the software for efficient resource management and workload balance. The second step is to tweak compiler options to investigate different auto optimization/vectorization techniques and performance benefits.

The core computation of BSB model is matrix vector multiplication as shown in equation (1). This is repeated for a maximum of 50 iterations or until the results converge whenever an input image is compared against a stored pattern. It is worth to note that the Intel Math Kernel Library (MKL) provides an optimized parallel implementation for matrix-vector multiplication, where the original matrix and vector are segmented and loaded to different cores for distributed processing. The results will be merged at the end. However, the performance of such fine-grained workload partition and parallelization is severely limited by Amdahl's law. Unless the size of the matrix and vector is sufficiently large, the performance gain from parallel computing is not enough to offset the overhead of communication and synchronization [26]. Furthermore, to get best performance, MKL allocates the maximum resources i.e. all the cores, for one matrix-vector operation. Hence we have to serialize the bottom layer of ITRS and run the pattern-matching tasks one by one. It was observed that such globally serial and locally parallel (GSLP) implementation is not efficient for ITRS.

In contrast to GSLP, we adopt a globally parallel and locally serial approach. OpenMP threads and pthreads are created and distributed across Xeon Phi to handle multiple pattern-matching tasks independently. They are referred as *solver threads*. All threads run in parallel. Their

synchronization is handled by thread safe blocking queues, which have critical sections defined for accessing the queue and blocks the thread if the queue is empty. This allows the architecture to be inherently load balancing as each computation thread can pick up workload from the queues whenever it finishes processing the current task. The compute threads have data exchange only with the thread-safe queue. The communication with the rest of the ITRS system, which runs on the host CPU, is handled by another thread. By decoupling the compute thread from MPI communication, we keep them busy for maximum duration.

7.1 Software architecture optimizations

The first step of optimization is to the find the best software architecture for efficient resource management and workload balancing. For any combination of input image and stored pattern for comparison, a pattern-matching task is created. The set of pattern-matching tasks for all of the 96 input images forms the workload. Based on how the workload is partitioned and distributed, three different resource management schemes are tested and their performance is compared.

7.1.1 Multiple comparison patterns to multiple OpenMP threads (MPMT)

In this architecture 244 solver threads are created using OpenMP. This is the number of logical cores available on the coprocessor card. Any pattern-matching job can be assigned to any of the available solver threads. Figure 20 shows how the workload is created and assigned.

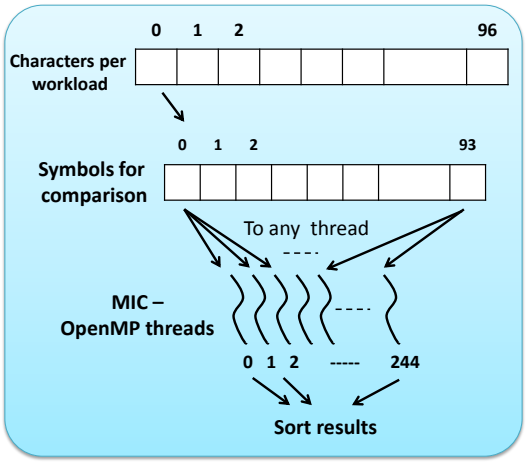


Figure 20: MPMT architecture

Because the threads can work on any available job at any time, this approach has excellent load balancing ability. However, the BSB models for different patterns have different weight matrices. Due to the limited cache size, every time a new pattern-matching job is started, a new weight matrix of the BSB model (corresponding to the pattern to be compared) must be shuttled/read into the target core's local cache. When 244 threads running simultaneously, large amount of data transfer is created, which causes bus contention. Explicit data management to preserve data locality can improve the performance significantly. Using this resource management scheme, it takes 18.41 seconds to process 96 input images with 30x30 resolution. The performance analysis from VTune Amplifier confirms that there were huge memory stalls in the compute section of the solver thread. This indicates that the performance of the pattern-matching layer is bounded by memory performance.

7.1.2 Specific comparison pattern to specific pthread (SPST)

To get finer control over data, thread creation and destruction, pthreads are used instead of OpenMP threads as shown in Figure 21. A set of 93 pthreads are created during initialization

and destroyed only when the ITRS terminates. There are 3 other threads, which take care of MPI communication and cleanup. We also divide the entire workload into 93 sets. Each set of workload contains pattern-matching tasks between all input images and one of the 93 stored patterns. A specific set of workload is assigned to a specific pthread. Because each pthread always compares the input with the same stored pattern, the weight matrix of the corresponding BSB model can stay in the cache. Therefore, the data contention problem in MPMT is relieved.

Since the solver threads are alive for the entire duration of the program and each thread works on a specific symbol, there is better utilization of cache. Compared to MPMT, the work distribution of SPST may not be uniform but the performance gained due to efficient memory utilization outperforms load balancing overhead

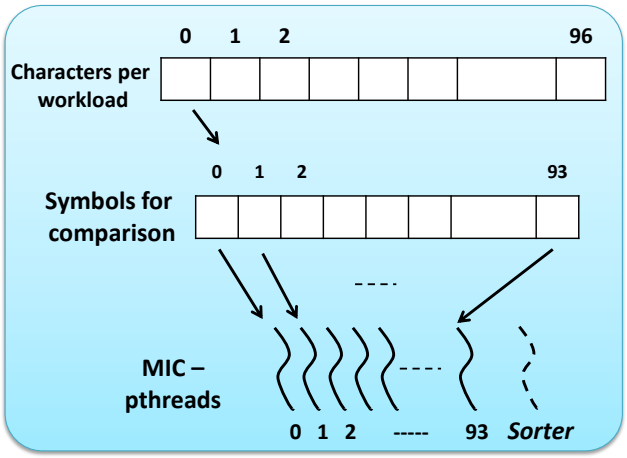


Figure 21: SPST architecture

Performance can also be tweaked by changing the thread affinity. Specifying affinity of a particular thread makes it run on the specified logical core. Three affinity settings were tried out as described below:

Compact affinity: Each thread was assigned to adjacent core. The run time was 18.5 seconds. This is not very ideal setting as every physical core (and its local cache) is shared by 4 pthreads.

Affinity for alternate logical cores: In this case each physical core will run no more than two threads. This is an improvement over the previous case because only 2 threads share a physical core and the cache. The runtime was 11.3 seconds.

Scatter affinity: By default the micro OS running on the coprocessor scatters the threads among the 244 logical cores. This is the best configuration, as it tries to minimize cache sharing among threads. The runtime in this case was 10.37 seconds.

Performance analysis by VTune Amplifier shows that the optimized solver code was not getting steady stream of data due to sustained memory bandwidth limitations and smaller local cache size for the required data. We were able to achieve 176.58 GB/s memory bandwidth utilization.

7.1.3 Specific pattern to specific pair of pthreads (SP2T)

The architecture in Figure 22 was developed to improve weight matrix data retention in cache. It is similar to SPST, however, each weight matrix was split into halves and distributed to a solver thread and its companion thread. While issuing characters for comparison a duplicate copy was also issued to the companion thread. Hence one workload now needs 93*2=186 threads as shown in Figure 22.

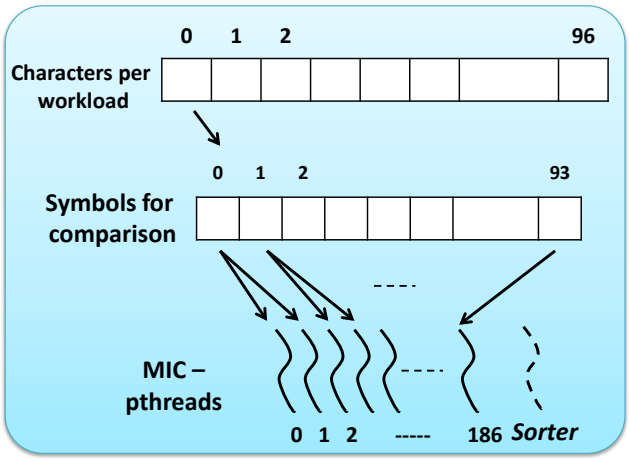


Figure 22: SP2T architecture

Although the core computation in this case took the same time as SPST, new overhead for syncing the computation between the thread pairs is added. The runtime had now increased to 13.92 seconds.

7.2 Compiler based optimizations

Compiler switches and corresponding pragmas, language extensions and appropriate coding styles were employed to assist in auto optimization, based on the guidelines provided by the Intel compiler. Loop unrolling, vectorization, prefetching, streaming stores and Inter Procedural optimization (IPO) were evaluated.

Streaming stores and IPO had limited boost in performance due to the nature of the BSB algorithm.

7.2.1 Loop unrolling

This is a technique where the program's speed is increased by converting loops to linear code and performing optimizations on this code. Some optimizations that can be performed are reducing or removing instructions that control the loop, simplifying or reducing some pointer

arithmetic, reducing the number of looping condition tests and number of iterations. It also reduces branch penalties and opens up opportunities for data prefetching which is crucial in hiding latencies for memory intensive operations etc.

Pragmas were inserted in the code and compiler switches were used to enable loop unrolling. Since the software architecture was already refined the compiler generated optimized loops without the need for additional guidance for loop unrolling.

7.2.2 Streaming stores

Streaming stores are a set of instructions which bypass the cache and directly update the memory location. With this optimization normal loops saw a drop in runtime of about 6 seconds, but blocked loops (loops where data is partitioned for cache line) didn't show any significant change. This is because of the nature of data access required by the BSB algorithm, where higher number of stream reads are required compared to the number of stream writes.

7.2.3 Inter Procedural optimization (IPO)

IPO is a collection of compiler techniques used to improve performance of frequently used functions. It analyzes the entire program or a single block of code and tries to reduce or eliminate duplicate calculations, inefficient use of memory, and to simplify iterative sequences such as loops.

Our case didn't show any significant change in run time as the main computation kernel is simplified and vectorized.

7.2.4 Prefetching

The coprocessor does not have out of order execution but it incorporates prefetch techniques to keep the pipeline full. It is accomplished by issuing prefetch instructions interleaved between other instructions before the actual need for the specified data/instruction. These instructions don't stall the processor and the data/instructions will be available ready in cache by the time they are actually needed.

The coprocessor supports both hardware prefetch and software prefetch. In our case it relies more on software prefetching than on hardware prefetch. Hardware prefetching is enabled by default. The BSB algorithm did not benefit from the hardware prefetcher due to the nature of data access pattern required by the algorithm. However software prefetching had significant impact on the runtime and is enabled by default. For the case of 30x30 character BSB run needed about 16 seconds without software prefetching and after enabling, it took about 10 seconds for the same run.

An experiment was carried out by manually adding prefetch intrinsics and hints for prefetch distances to C++ code. In comparison the compiler optimized code provided better performance as it was able to compute optimum prefetch distances. As an observation in this particular case; manually adding prefetch is probably best suited if the coding style is at assembly level or for intrinsic heavy coding format.

7.2.5 Vectorization

The Xeon Phi compiler performs vectoization, which converts scalar operations (operations on one set of data) to vector operations (same operation is performed on multiple data). One

vector instruction operates on multiple operands hence significantly reducing the effective runtime compared to scalar implementation.

The coprocessor has vector processing units with 512 bit vector registers. These VPUs help in greatly reducing the code runtime. In fact vectorization provided the largest boost to the overall performance. All the results presented in this section are obtained with vectorized code.

Vectorization was achieved by recoding computation loops in a specific pattern [22], along with the use of language extensions (restrict) specific to the Intel compiler and respective compiler switches. Also the data had to be memory aligned to benefit from these optimizations. The code was fine-tuned and the required level of optimization achieved was confirmed through specific compiler reports.

8 Results and Analysis

8.1 Overall performance optimization

The Figure 23 shown below, provides performance snapshot of different implementation milestones. The performance achieved for best case scenario in terms of Giga FLOPs is shown in Figure 24.

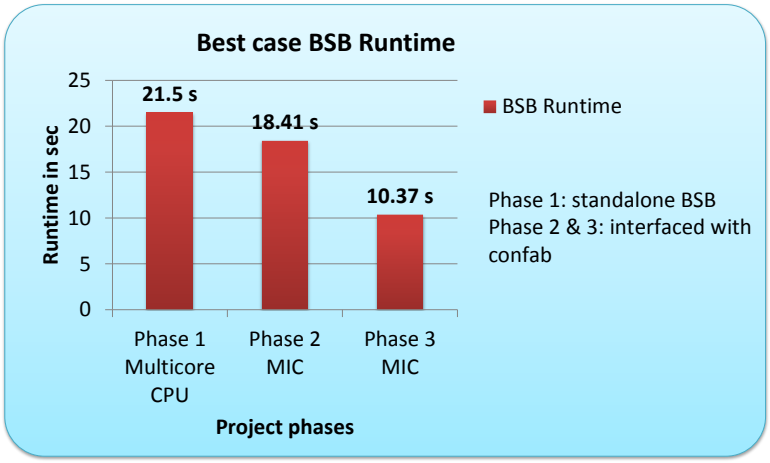


Figure 23: Phase by phase runtime reduction

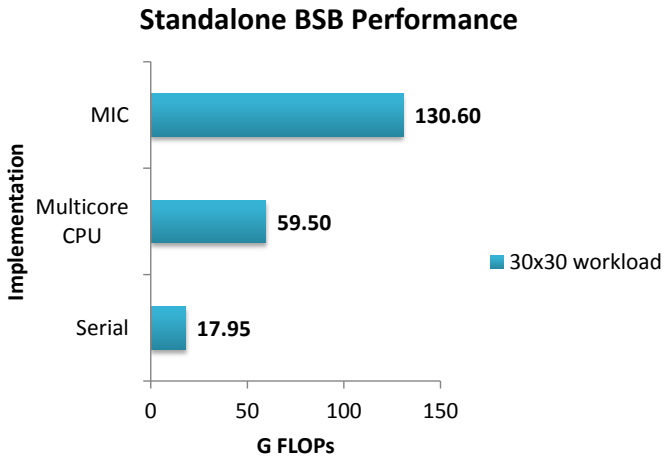


Figure 24: FLOPs achieved

8.2 Results

After applying the SPST resource management with scatter affinity, and with the help of compiler based optimization options, we were able to optimize the pattern-matching layer for MIC architecture and achieved 1.8x performance gain on Intel's Xeon Phi coprocessor over MPMT as shown in Figure 25. The runtimes plotted are for one workload of 96 characters at 30x30 resolution.

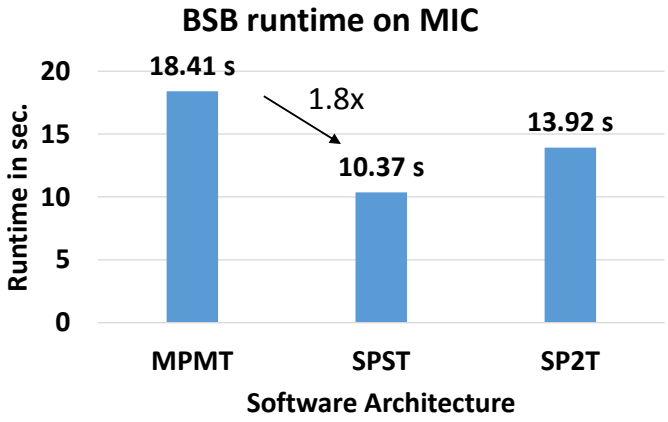


Figure 25: BSB Runtime Optimization

For fair comparison, the same pattern-matching layer is also implemented on a standalone CPU and IBM Cell processor based PlayStation 3® setup. The CPU used in this experiment has 16 physical cores, each supporting 2-way simultaneous multi-threading. Each core has 512KB L1 cache, 2MB L2 cache and 20MB L3 cache. Since there are more threads than logic cores, the workload balancing is done by OS. Each PS3 processor has 6 Synergetic Processing Elements (SPE) and one PowerPC processor. Each SPE handles one SPST threads.

We consider the performance of the serial version of BSB algorithm running on CPU as our base reference and set its performance to 1. Figure 26 gives the normalized performance of the pattern matching layer implemented on CPU, Xeon Phi and PS3. Because the same software architecture is implemented, the comparison measures the performance gained by upgrading the hardware to the MIC architectures. There is no Cell processor implementation data for 30x30 resolution case, as it was not feasible to run at this resolution due to limited memory. As we can see, Xeon Phi is able to provide 1.35x performance gain over the PS3 for 15x15 images and 1.94x performance gain over optimized CPU implementation for 30x30 images.

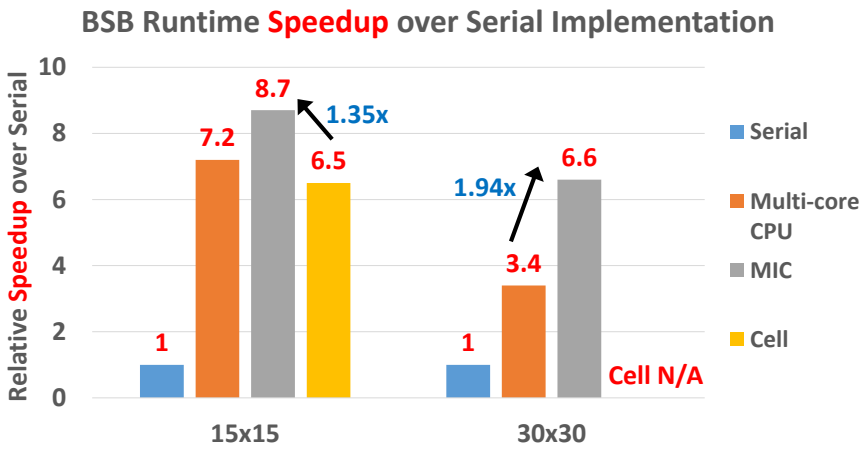


Figure 26: Runtime comparison for standalone case

Figure 27 shows the normalized performance comparison considering the communication interface with the rest of the ITRS. Again, Xeon Phi is able to provide 1.46x performance gain over the PS3 for 15x15 images and 1.9x gain over the optimized CPU implementation for 30x30 images.

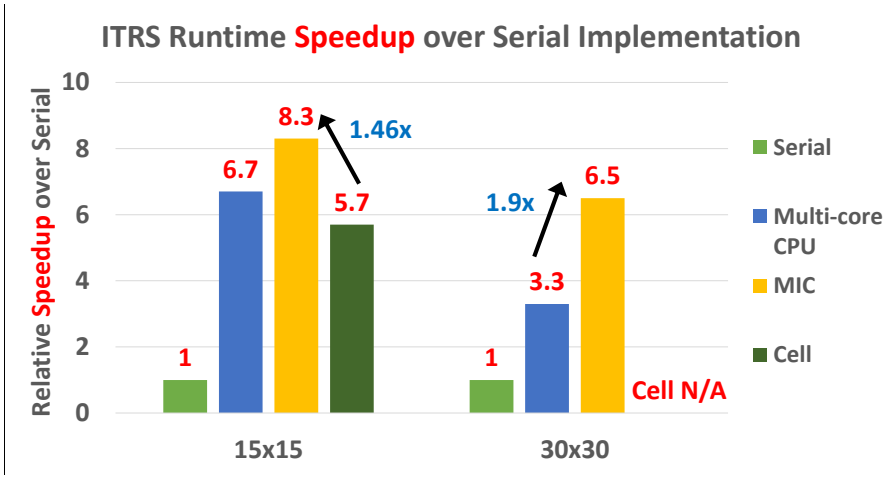


Figure 27: Runtime comparison with confab

8.3 Analysis

The pattern matching workload for each input character image is a matrix multiplication, for 30x30 character it is [1024 x 1024] * [1024 x 1]. Each element in these matrices is a float data, hence the size of the weight matrix is 4MB and the size of the input vector is 4KB. The result of the matrix vector multiplication generates 4KB of new data. For each input image, the number of times this multiplication is performed is: 50*93 = 4650, where 50 is the maximum number of iterations allowed for convergence. The above number of iterations is repeated for all the characters in the workload i.e. 96 times.

Total data requirement per iteration is ~4MB. Total cache on the coprocessor is about 30 MB. This cache is coherent and can be accessible through any core. Each core has 32 KB L1 cache and 512 KB L2 caches.

The data required per iteration is significantly greater than the per core cache size hence there is memory spill over per iteration. This is clearly the bottleneck which is holding back the overall performance. The nature of core compute part of the algorithm is like stream read. This

application is data intensive and the achieved bandwidth is 176.58 GB/s (peak) which is almost same as Stream Memory Benchmark which peaks at 181 GB/s [27].

The BSB runs natively on the coprocessor hence the Xeon host is dedicated solely for confabulation. This is especially beneficent as the BSB can run independently by receiving workload requests and send results through MPI without stalling any other module. Hence as many BSBs can run as the system can support.

The software architecture where a specific symbol is issued to a specific pthread as shown in Figure 21 is found to be ideal for MIC architecture. The scaling capabilities are described based on this software architecture. This architecture is very flexible and allows for testing the behavior in terms of scaling at the workload level with in the Xeon Phi coprocessor. For one workload configuration 3 + 93 + 1 = 97 threads are required. The number of threads for two simultaneous workload configuration is 3 + (93 + 1) * 2 = 191 threads. Going beyond this is not advisable as number of threads will exceed the available hardware resources.

The BSB also scales at the node level and cluster level along with ITRS as shown in Figure 28. The BSBs communicate directly with the host Xeon processor and not with each other. This kind of partitioning helps in maintaining simplicity and low MPI communication delays. Using multiple coprocessor cards in a single node provides linear performance scaling of BSB. The same scaling benefit can be achieved at the cluster level as well.

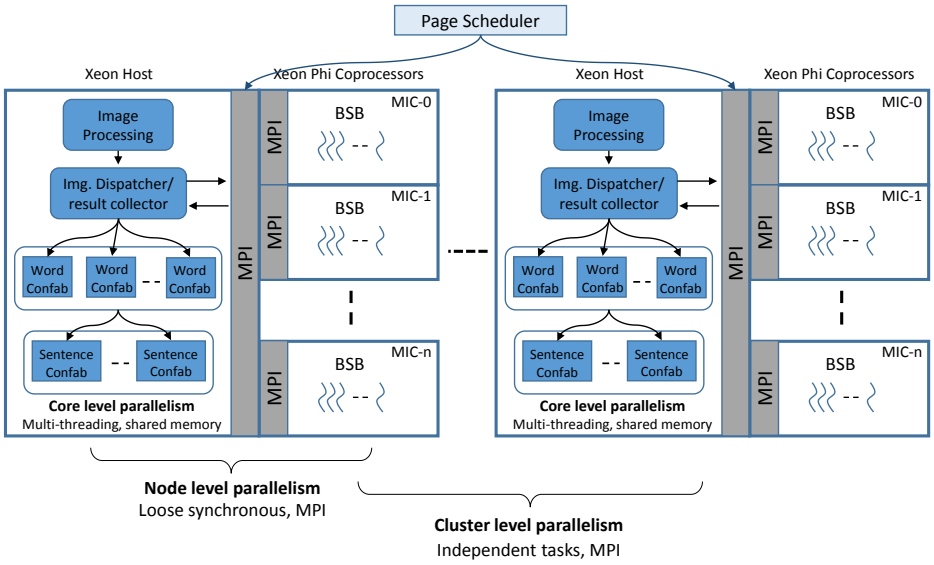
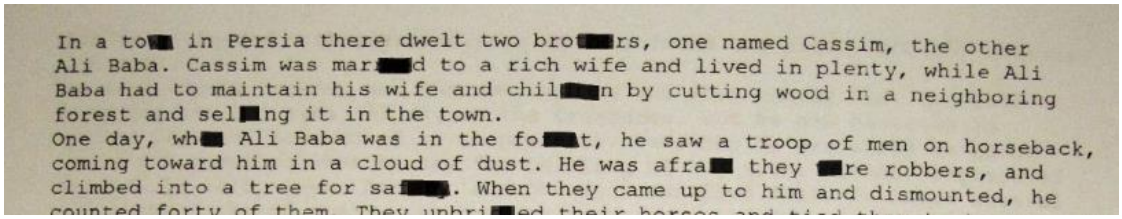


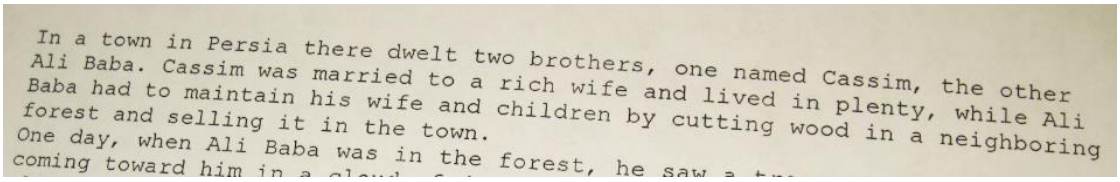
Figure 28: Multi-level scaling of hybrid ITRS architecture

8.4 Sample run for ITRS

8.4.1 Sample inputs



In a town in Persia there dwelt two brothers, one named Cassim, the other Ali Baba. Cassim was married to a rich wife and lived in plenty, while Ali Baba had to maintain his wife and children by cutting wood in a neighboring forest and selling it in the town.
One day, when Ali Baba was in the forest, he saw a troop of men on horseback, coming toward him in a cloud of dust. He was afraid they were robbers, and



In a town in Persia there dwelt two brothers, one named Cassim, the other Ali Baba. Cassim was married to a rich wife and lived in plenty, while Ali Baba had to maintain his wife and children by cutting wood in a neighboring forest and selling it in the town.
One day, when Ali Baba was in the forest, he saw a
coming toward him in a

8.4.2 Output for 15x15 characters

```
================== BSB performance on MIC ======================
 Total number of workloads processed 2, with each having 96 chars, using 1 Recall Solvers
  -------- Recall Solver: 0
   Total runtime MIN: 0.5106    |         per workload runtime in Recall Solver MIN: 0.2553
   Total runtime AVG: 0.9018    |         per workload runtime in Recall Solver AVG: 0.4509
   Total runtime MAX: 1.0122    |         per workload runtime in Recall Solver MAX: 0.5061

   ------ Effective runtime per workload in BSB
    per workload (without MPI) MIN: 0.2553
    per workload (without MPI) AVG: 0.4509
    per workload (without MPI) MAX: 0.5061



================== Sentence confabulation results ==================

--Sentence 0 results
WORD CONFAB:
(in ih ln lh 1n 1h )(a d h )(tcwn tcnn town towh tonn tonh bcwn bcnn bown bowh bonn bonh fcwn
fcnn f ...
SENTENCE CONFAB --------------- in a tcwn in persia there dwelt two brothers  ----------


--Sentence 1 results
WORD CONFAB:
(cne cnc cno che chc cho one onc ono ohe ohc oho )(named namea namcd namca namod namoa naned
nanea n ...
SENTENCE CONFAB --------------- one named cassim  ----------



================== Sentence confabulation runtime ==================

--Per sentence confabulation runtime including word confab: 6.2493s
```

8.4.3 Output for 30x30 characters

```
================== BSB performance on MIC ======================
 Total number of workloads processed 1, with each having 96 chars, using 1 Recall Solvers
  -------- Recall Solver: 0
   Total runtime MIN: 5.7198    |         per workload runtime in Recall Solver MIN: 5.7198
   Total runtime AVG: 10.3190   |         per workload runtime in Recall Solver AVG: 10.3190
   Total runtime MAX: 11.1757   |         per workload runtime in Recall Solver MAX: 11.1757

   ------ Effective runtime per workload in BSB
    per workload (without MPI) MIN: 5.7198
    per workload (without MPI) AVG: 10.3190
    per workload (without MPI) MAX: 11.1757



================== Sentence confabulation results ==================

--Sentence 0 results
WORD CONFAB:
(ln lh in ih )(a d j )(town towh tonn tonh bown bowh bonn bonh fown fowh fonn fonh )(in ih ln lh
)(p ...
SENTENCE CONFAB ---------------- in a town in persia there dweit two brothers  ----------


--Sentence 1 results
WORD CONFAB:
(one onc ono ohe ohc oho )(named nameu namcd namcu namod namou naded nadeu nadcu nadod nadou
naned n ...
SENTENCE CONFAB ---------------- one named cassim  ----------



================== Sentence confabulation runtime ==================

--Per sentence confabulation runtime including word confab: 6.2873s
```

9 Conclusion and future work

9.1 Conclusion

We started off with a goal of upgrading the processing capability and accelerating BSB by having an efficient and optimized platform which can scale up to a cluster level. We parallelized and optimized the serial version of BSB for Xeon Phi coprocessor. During optimization we explored several avenues on the software architecture side and tweaked auto-optimization features available. We explored the effectiveness of using OpenMP and pthreads for the BSB algorithm. Both 15x15 and 30x30 resolution images were experimented on and found that 30x30 case is now feasible. Overall we were able to achieve a speed up of ~2x with our ITRS hybrid Xeon – Xeon Phi coprocessor implementation. This architecture is scalable at the core level, node level and at the cluster level. Every BSB (coprocessor) added to the system provides linear scaling in overall performance of all BSBs combined.

9.2 Future work

Our future work on this problem is to optimize the architecture for memory bandwidth. Also we will be reformulating the algorithm such that the computation can be split on multiple cores and span multiple iterations, to improve data retention on the coprocessor.

10 Appendix

10.1 OpenMP

It is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism. OpenMP is a short for Open Multi-Processing. It is comprised of three primary API components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

The main purpose of OpenMP is to provide a standard among a variety of shared memory architectures/platforms. It is simple and limited set of directives for programming shared memory machines. It provides capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach. The standard provides the ability to implement both coarse-grain and fine-grain parallelism utilizing fork join threading model. This API is specified for C/C++ and Fortran.

OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory – Uniform Memory Access (UMA) or Non Uniform Memory Access (NUMA) as shown in Figure 29.

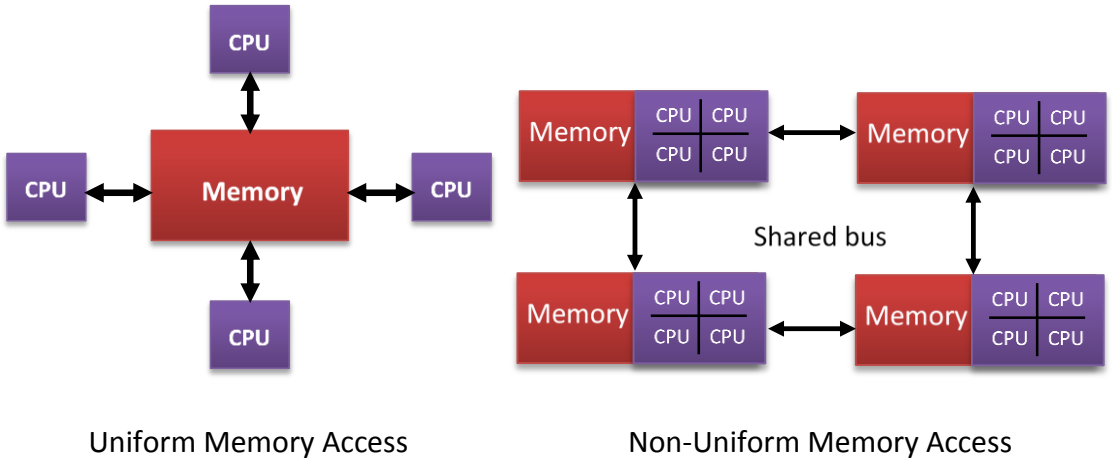


Figure 29: Shared memory model

Source: https://computing.llnl.gov/tutorials/openMP

10.2 MPI

MPI stands for Message Passing Interface. It is a standard established for portable, efficient and flexible message passing application. It primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process. The standard allows for communication in case of shared memory or with distributed memory as well as with hybrid memory architectures as shown in Figure 30.

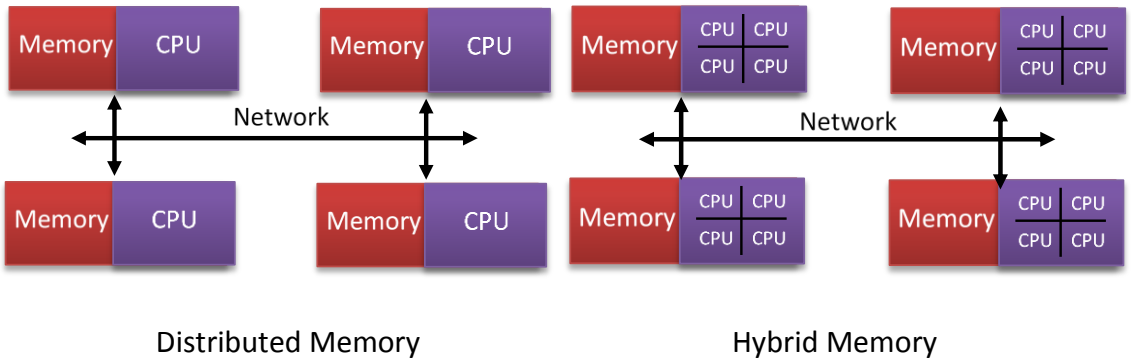


Figure 30: Memory architecture

Source: https://computing.llnl.gov/tutorials/mpi/

MPI is a common standard and is widely used on HPC platforms. It provides the ability to abstract the underlying communication technologies and hence is very portable. All parallelism is explicit and programmer specified.

11 References

[1] Qinru Qiu; Qing Wu; Bishop, M.; Pino, R.E.; Linderman, R.W., "A Parallel Neuromorphic Text Recognition System and Its Implementation on a Heterogeneous High-Performance Computing Cluster,"*Computers, IEEE Transactions on* , vol.62, no.5, pp.886,899, May 2013 doi: 10.1109/TC.2012.50

[2] R. Wray, C. Lebiere, P. Weinstein, K. Jha, J. Springer, T. Belding, B. Best, and V. Parunak, "Towards a Complete, Multilevel Cognitive Architecture," *Proc. of the International Conference for Cognitive Modeling*, 2007.

[3] R. S. Swenson, "*Review of clinical and functional neuroscience*," Educational Review Manual in Neurology, Castle Connolly Graduate Medical Publishing, 2006.

[4] J. A. Anderson, "*An Introduction to Neural Networks*," The MIT Press, 1995.

[5] George Chrysos, Intel Corporation Intel® Xeon Phi™ Coprocessor - the Architecture http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

[6] Intel® Xeon Phi™ Coprocessor System Software Developers Guide http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-system-software-developers-guide

[7] J. Park, Y. Park, "An Optimization Approach to Design of Generalized BSB Neural Associative Memories," *Neural Computation, MIT Press Journals*, Vol. 12, No. 6, Jun. 2000, pp. 1449-1462.

[8] Y. Park, "Optimal and Robust Design of Brain-State-in-a-Box Neural Associative Memories," *Neural Networks, Elsevier*, Volume 23, Issue 2, Mar. 2010, pp. 210-218.

[9] Schultz, “Collective recall via the brain-state-in-a-box network,” *Neural Networks, IEEE Transactions on*, vol. 4, no. 4, pp. 580–587, 1993.

[10] S. Mori, C.Y. Suen, and K. Yamamoto, “Historical Review of OCR Research and Development,” *Proc. IEEE*, vol. 80, no. 7, pp. 1029-1058, July 1992.

[11] Jianchang Mao, "A case study on bagging, boosting and basic ensembles of neural networks for OCR," *Neural Networks Proceedings, 1998. IEEE World Congress on Computational Intelligence. The 1998 IEEE International Joint Conference on* , vol.3, no., pp.1828,1833 vol.3, 4-9 May 1998

[12] Blando, L.R.; Kanai, J.; Nartker, T.A., "Prediction of OCR accuracy using simple image features," *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on* , vol.1, no., pp.319,322 vol.1, 14-16 Aug 1995

[13] Peng Ye; Doermann, D., "Learning features for predicting OCR accuracy," *Pattern Recognition (ICPR), 2012 21st International Conference on* , vol., no., pp.3204,3207, 11-15 Nov. 2012

[14] Qing Wu; Mukre, P.; Linderman, Richard; Renz, T.; Burns, D.; Moore, M.; Qinru Qiu, "Performance optimization for pattern recognition using associative neural memory," *Multimedia and Expo, 2008 IEEE International Conference on* , vol., no., pp.1,4, June 23 2008-April 26 2008

[15] Taha, T.M.; Yalamanchili, P.; Bhuiyan, M.A.; Jalasutram, R.; Mohan, S.K., "Parallelizing two classes of neuromorphic models on the Cell multicore architecture," *Neural Networks, 2009. IJCNN 2009. International Joint Conference on* , vol., no., pp.3046,3053, 14-19 June 2009

[16] Miao Hu; Hai Li; Qing Wu; Rose, G.S.; Yiran Chen, "Memristor crossbar based hardware realization of BSB recall function," *Neural Networks (IJCNN), The 2012 International Joint Conference on* , vol., no., pp.1,7, 10-15 June 2012

[17] Honghoon Jang; Anjin Park; Keechul Jung, "Neural Network Implementation Using CUDA and OpenMP," *Digital Image Computing: Techniques and Applications (DICTA)*, 2008 , vol., no., pp.155,161, 1-3 Dec. 2008

[18] Billconan and Kavinguy. A neural network on gpu http://www.codeproject.com/Articles/24361/A-Neural-Network-on-GPU

[19] Nere, A.; Hashmi, A.; Lipasti, M., "Profiling Heterogeneous Multi-GPU Systems to Accelerate Cortically Inspired Learning Algorithms," *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International* , vol., no., pp.906,920, 16-20 May 2011

[20] Diaz, J.; Munoz-Caro, C.; Nino, A., "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era," *Parallel and Distributed Systems, IEEE Transactions on* , vol.23, no.8, pp.1369,1386, Aug. 2012

[21] Xinmin Tian; Saito, H.; Preis, S.V.; Garcia, E.N.; Kozhukhov, S.S.; Masten, M.; Cherkasov, A.G.; Panchenko, N., "Practical SIMD Vectorization Techniques for Intel® Xeon Phi Coprocessors," *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* , vol., no., pp.1149,1158, 20-24 May 2013

[22] David Mackay, Optimization and Performance Tuning for Intel® Xeon Phi™ Coprocessors - Part 1: Optimization Essentials http://software.intel.com/en-

us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization

[23] Shannon Cepeda, Optimization and Performance Tuning for Intel® Xeon Phi™ Coprocessors, Part 2: Understanding and Using Hardware Events http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding

[24] Gao Tao; Lu Yutong; Suo Guang, "Using MIC to Accelerate a Typical Data-Intensive Application: The Breadth-first Search," *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* , vol., no., pp.1117,1125, 20-24 May 2013

[25] Turchenko, V.; Bosilca, G.; Bouteiller, A.; Dongarra, J., "Efficient parallelization of batch pattern training algorithm on many-core and cluster architectures," *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2013 IEEE 7th International Conference on* , vol.02, no., pp.692,698, 12-14 Sept. 2013

[26] Intel® Math Kernel Library http://software.intel.com/en-us/intel-mkl

[27] Karthik Raman Optimizing Memory Bandwidth on Stream Triad http://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad

[28] Blaise Barney, OpenMP https://computing.llnl.gov/tutorials/openMP

[29] Blaise Barney, Message Passing Interface (MPI) https://computing.llnl.gov/tutorials/mpi/

12 Vita

Education

R.N.S INSTITUTE OF TECHNOLOGY, Bangalore, India (V T U)
Bachelor of Engineering in Electronics and Communication

Professional Experience

INTEL CORPORATION, Santa Clara, CA
Intern [Jan 2013 to July 2013]

- Explored and implemented various software architectures to optimize the performance of BSB (brain state in a box) algorithm used for character pattern recognition on Intel Xeon Phi coprocessor.

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING, Syracuse University, Syracuse, NY
Research Assistant

- Document Image Parsing and Understanding using Neuromorphic Architecture [Jan 2012 to Dec 2012], [Aug 2013 to present]
- Surface EMG signal processing on FPGA [Feb 2011 to July 2011]
- Researched building a system that is immune to malware [Aug 2009 to Dec 2009]

Teaching Assistant

- Object Oriented Design course [CSE 687 - Spring 2010], [CSE 283 - Fall 2011]

IWAVE SYSTEMS TECHNOLOGIES PVT LTD. (www.iwavesystems.com) Bangalore, India
Engineer (Member Technical – Hardware) [May 2007 to July 2008]

- Develop, test and maintain storage IP cores

Publication / Presentations

- Poster on “Brain Inspired Information Association on Hardware” at DATE 2013 conference
- Paper on “A novel approach for simulation, measurement and representation of surface EMG (sEMG) signals” at IEEE ASILOMAR 2011 conference
- Poster on FPGA based circuit simulation using wavelets on Nunan Lecture And Research Day at Syracuse University (2010), judged as a runner-up
- Poster on Digital Synthesizer/Mixer on Nunan Lecture And Research Day at Syracuse University (2009)
- Paper on improving MPEG system to accommodate 3D perspective
- Verilog tutorial on behalf of GROVE (Group of VLSI Enthusiasts) with hands on session